DEUCE Interpretive Programs

by C. Robinson

Summary: This paper describes the principal features of (i) The *General Interpretive Program*, (ii) The *Tabular Interpretive Program*, and (iii) *Alphacode*, which are the interpretive programs which have been most extensively used in solving problems on DEUCE. The characteristics of these three schemes are compared and contrasted.

1 INTRODUCTION

In parallel with the organization of large libraries of programs and subroutines for a computer, there will usually be a demand for interpretive or translation routines designed to speed up programming for the "one-off" kind of job. The need for interpretive schemes will arise particularly in scientific or engineering research establishments, where only a fraction of the computing needs can be met by library routines, and where so much work is experimental. It is not surprising, therefore, that a number of interpretive schemes have been evolved for use on DEUCE, a computer which, so far at least, has been used mainly for the solution of scientific and engineering problems. The three schemes which have been most used are discussed here, and their mutually complementary features are apparent. It seems to be generally the case that, short of the direct use of formulae to define the solution to a problem, a "three address plus function" code is the best liked from the user's point of view. All the schemes mentioned here are of this type.

2 THE GENERAL INTERPRETIVE PROGRAM

This program, developed over three years ago at the National Physical Laboratory, has been and is very extensively used. That it has not been superseded is due to its remarkable flexibility and to the fact that, in general, it sacrifices so little in machine speed. The program accepts instruction words composed of 4 parts: a, b, c and r, where the value of r defines a function, and a, b and c are parameters (generally addresses of data) required in the execution of the order. The user's task is therefore to break down his problem into a series of "3 address plus function" codewords, where the functions available are more numerous and more comprehensive than those in a normal machine code.

The main feature of the scheme is that the functions specified by the operation number r are not unique and are selected by the programmer himself from a library. Thus, in preparing a sequence of orders for this scheme, the programmer decides which functions he needs, and then selects the appropriate programs from the library. The fact that the library of available functions has grown, and continues to grow, is one of the reasons why the scheme has not been out-dated.

The separate library programs which a programmer may use are called *bricks*, and any DEUCE program which satisfies a few simple conditions may be used as a standard brick with this scheme. The conditions are on how it starts and finishes—for if it requires any parameters they must be accepted from the stores in which the Interpretive Program (usually referred to as *G.I.P.*) plants them, and it must clearly end in a standard way so that G.I.P. can resume control. The high-speed store on DEUCE is large enough for most bricks, but a large brick may be split into several sections. In such cases each section but the last is arranged to leave a negative number in a particular store, and this is sufficient to ensure that G.I.P. brings down the various sections in turn. Thus a multi-section brick does not involve the interpretive programmer in any extra work; he merely writes one instruction word a, b, c, r and this causes all the sections of brick r to be obeyed in turn.

Having decided what bricks are required for a particular job, and having made the program of codewords, the operator then provides DEUCE with G.I.P., followed by the various bricks, followed by the codewords. G.I.P. and the bricks are packed at the "top end" (i.e. in high-numbered tracks) of the magnetic drum, leaving a large consecutive store at the "bottom end" (i.e. in low-numbered tracks) to be used for storing data. G.I.P. examines the first codeword, selects the rth brick of those currently stored, fetches that brick into the high-speed store, plants a, b and c in a convenient store where they will be picked up by the brick, and finally enters the brick at its first instruction. At the conclusion of the brick, G.I.P. regains control, the next codeword is examined, and the process is continued.

As described so far, G.I.P. is merely linking together the various self-contained sections of a program. However, seventeen values of r are set aside for instruction modification, discriminations, counting, and other "housekeeping" operations so that all the requirements of a full machine order code are met. In codewords using these special values of r, the quantities a, b and c usually refer to the numbers of other codewords (which are numbered serially). Discriminations thus take the form "Jump to codeword number b if the contents of codeword a are zero, otherwise jump to codeword number c." Such a codeword implies that one codeword (in this case number a) is being used as a counter rather than as a codeword to be obeyed. There are a variety of discriminations provided, as well as an unconditional jump. Others among these special values of r cater for automatic instruction modification (modify a, b or c automatically after the codeword has been obeyed), and provide facilities for reading in new bricks

and either writing them over bricks already in store and which are no longer needed, or obeying them directly in the high-speed store. The latter facility is particularly helpful when one is embarrassed for drum storage space and there are one or more bricks which are obeyed only once in a program. These may as well be read and obeyed directly, once and for all, and so save the storage space which they would otherwise need on the drum.

To use G.I.P., therefore, a programmer does not have to possess any detailed knowledge of DEUCE programming, providing he can achieve his object by using existing bricks. He will have to count up how many of the 256 tracks on the drum will be needed by G.I.P. (which accounts for 21) and all the bricks he needs for his particular operation. The remaining tracks, which are always the lowest track numbers, are available for his data, and it will be necessary to know how the various bricks being used expect to find and store their data. Since virtually all bricks are built to accept and store data in a standard manner, the programmer's problem is to apportion the data tracks correctly. There are, nevertheless, features which a DEUCE programmer may use, since any codeword, as an alternative to the standard *a*, *b*, *c*, *r* form, may be replaced by a normal DEUCE instruction. The interpretive program will detect such an instruction (by the presence of a digit normally spare), and obey it as required. For instance, when one needs to read a single number into the machine, it might be easier to use the DEUCE instruction facility rather than to call a brick to perform such a trivial operation.

The program provides generous testing facilities in the way expected of a conventional machine order code. There is the equivalent of a "stop key" which, when on, will cause the machine to stop on every G.I.P. instruction, and display it on lights before it is obeyed. Similarly, one can force G.I.P. to stop on a specific instruction and display it, by setting the serial number of that instruction on the keys—this facility being of use when an instruction is being examined each time it is obeyed in a loop. There are also convenient facilities for making the machine accept an instruction to replace that which it is about to obey, and to restore control or take a post-mortem in the case of a brick not behaving as expected.

3 THE USE OF THE GENERAL INTERPRETIVE PROGRAM

The majority of the bricks available for use with G.I.P. are for linear-algebra operations, and as a result there has arisen a widespread belief that the scheme is restricted to matrix operations. This is not so, although the scheme has been used extensively for linear algebra and, in fact, many problems which do not, at first sight, appear to be linear-algebra problems, have been solved using standard matrix bricks. Gilmour (1958) discussed a program to compute relationships between tractive effort, speed, efficiency and motor current for traction motors, which was solved by linear algebra, using, in particular, vector and matrix notation for recording the data from three-dimensional curves. Another example is a program, recently completed, to calculate the most efficient way in which to meet a given load on an electricity supply system, where a weekly load curve is supplied, and where given amounts of thermal, atomic, hydroelectric and pumped-storage generation are available at the appropriate economic rates. In this problem the various thermal stations had their own costs of generation, only a limited amount of water was available for hydroelectric generation at the most economic times of the week, and only a limited capacity of pumped-storage generating and pumping equipment was to be worked at a given efficiency. The problem was to compute the cost of running the system in the most efficient way. Treating the given load as a series of, say, half-hourly readings, the week's load curve was stored as a 1 by 336 vector, and the whole operation was reduced to vector algebra making use of existing bricks. It was found necessary to make one or two "special" bricks to do this particular job. This emphasizes the point that there are many problems, which would involve long and tedious programming, which can be solved in almost the same machine time but with much less programming time, by using G.I.P. and making one or two bricks to cover operations which are peculiar to the problem being solved. A number of standard statistical problems come into this category.

Since each function being carried out under the control of G.I.P. may be a complete DEUCE program in itself, which will be obeyed at optimum speed and which may be operating on a large amount of data, the time spent in interpreting the codeword and fetching the brick may not be a significant amount. In fact it is almost one second. G.I.P. therefore shows up to its best advantage when it is required to operate upon large quantities of data in parallel, as typified by sizeable matrix operations. It is least efficient whenever the operations are trivial or only affect a small amount of data. It is to cope adequately with calculations on small amounts of data that other schemes, to be mentioned later, have been developed.

Particular features of standard bricks are the elaborate checks on arithmetic accuracy and the reliability of the operator or programmer. This standard of checking was set with the first batch of bricks, and has been followed by later programs. All matrices stored on the magnetic drum have the grand sum of all the elements stored with them and, whenever any brick makes reference to the matrix, the grand sum is checked. This is important, not so much as an arithmetical check, but as a check that the programmer has not unwittingly overwritten the "tail end" of a matrix (which will in general be stored on a series of consecutive tracks). Similarly checks on the compatibility of matrices in, for instance, addition or multiplication are incorporated, and such checks have proved themselves very worth while—particularly in complicated operations where a programmer may mistake the dimensions of his matrices. For example, operations such as extracting rows, columns or sub-matrices, or compounding rows or

columns, do admit of human errors in determination of the dimensions!

A convention which has been adopted is that, whenever a matrix is stored on the drum, not only is its grand sum stored with it for checking purposes, but the dimensions are also stored with it. In this way the necessity for the programmer to specify the dimensions is avoided—he merely has to specify the first of the (consecutive) drum tracks on which the matrix is stored. A consequence of this is that if a G.I.P. program is made to cope with a given maximum size of matrices (the restriction being storage space on the drum) then the same program, without any alteration, may be used for any matrices whose size does not exceed the given maximum. In nearly all bricks the numbers are stored in block-floating form, that is with all elements to the same number of binary places, and with the largest element shifted up to fill the storage register. This method successfully combines the speed and storage economy of fixed-point working with the flexibility and accuracy of floating-point operation.

As a simple (but not very useful) example we can consider the program to add two matrices together, i.e. to add together corresponding elements in two arrays of numbers. We shall need three bricks: one to read in the data, one to do the addition, and one to punch the answer. For this, G.I.P. would be followed by three bricks: Read Matrix, Add Matrix, and Punch Matrix, and the interpretive program would consist of 4 instructions:

a	*b*	*c*	*r*	
0	0	0	1	Obey brick No. 1 (i.e. read a matrix) and store it in track 0 onwards on the magnetic drum. The number in the *c* position indicates where the matrix is to be stored and, in this instruction, *a* and *b* have no use.
0	0	110	1	Read a matrix into track 110 onwards.
0	110	110	2	Obey brick No. 2 (i.e. add matrices) providing it with parameters 0, 110 and 110. The *add matrix* brick will interpret this as a request to add the matrix at track 0 onwards to that at track 110 onwards and put the result at track 110 onwards, overwriting the one originally there.
110	0	0	3	Obey brick No. 3 (i.e. punch matrix), the matrix to be punched being found at track 110 onwards.

This program would be valid for any matrices which occupy less than 110 tracks on the drum, i.e. which have less than 3,516 elements each. (There are 32 elements to a track, and four extra words are required to store the dimensions of the matrix, its number of binary places, and its grand sum.) The figure of 110 is not arbitrary, and has been arrived at as follows. G.I.P. uses 21 tracks, and the three bricks use 14 tracks, so 221 tracks are left for data. Not more than 2 matrices need to be stored simultaneously, and therefore we can afford 110 tracks for each.

With programming only requiring so little effort, and with so many problems needing bulk calculations in parallel on relatively large amounts of data, it will be readily appreciated why, at several DEUCE installations, over half the production time has been spent using G.I.P.

4 THE TABULAR INTERPRETIVE PROGRAM

The other two interpretive schemes which have been widely used were both devised primarily to satisfy the requirement of a machine code which would allow a scientist or engineer, not particularly interested in orthodox programming, to present his problem to DEUCE in the terms in which he was used to working. In particular he could make programs himself without ever having heard of a magnetic drum, a delay line, the binary scale, optimum coding, or even a punched card. The first of these programs, the *Tabular Interpretive Program*, was devised at Bristol Aero Engines, and is remarkable for its extreme simplicity. The potential user was accustomed to carrying out calculations on a large sheet of paper ruled into rows and columns, and was accustomed to calculations in which a desk-machine operator found a function of the numbers in one or more columns and wrote this down in another column. The interpretive program (referred to as T.I.P.), therefore set out to present DEUCE to the user as just such a sheet of paper, but with a built-in operator! The user is told that he has a large sheet of paper, ruled into 128 columns and 32 rows, and that orders to the machine consist of instructions to perform an operation on all the numbers in one or two columns and write the results in another column.

The instruction word is again of the form *a*, *b*, *c*, *r*, and, for example, the instruction

7 18 23 2

has the significance "Add the numbers in column 7 to those in column 18 and write the answers in column 23" (the number 2 being interpreted as "add"). Similarly, the instruction

101 0 127 8

has the significance "take the logarithms of the numbers in column 101 and list the results in the corresponding rows of column 127." The 8 here is interpreted as "logarithm," and the 0 has no significance. A number of functions are available, including all the common mathematical functions, reading the ordinates of a graph, and subsequently interpolating in the "graph." It is also possible to replace existing functions by special ones particularly suited to one kind of user. There is a store set aside for constants, and the instruction

7 N18 23 2

will cause the numbers in column 7 each to be added to the 18th constant, and the results written in the corresponding rows of column 23. If an asterisk is written beside any number, that number will have 1 added to

it after it has been obeyed each time; an instruction with $r = 18$ will re-set all asterisked instructions in the preceding loop back to their initial values. Facilities known as "branching" and "mixing" are available for discriminating on the numbers in a column, taking different lines of action on the various elements of the column as a result of the discrimination, and subsequently mixing the results together. For more complicated programs it is possible to make subroutines of codewords.

The overriding consideration in devising the scheme, however, has been the necessity to present computing to an engineer in terms in which he is used to thinking, and the outcome is a scheme which can be learned in an hour or two and practised with a programming manual which need not be more than one sheet of paper. The scheme can be looked upon rather as a special version of G.I.P., with a fixed order code and dealing in vector algebra only. The interpretation time (of the order of 0·25 sec) is less, however, and it demands no knowledge of DEUCE. There are generous testing facilities, mostly similar to those for G.I.P., but also including a facility for printing out the first number only of every column or of selected columns. This is a great asset in running through a problem for the first time.

5 ALPHACODE

It will be apparent that, whereas G.I.P. is particularly suited for bulk calculations in parallel, and T.I.P. for a smaller number of calculations in parallel, neither scheme will show up to best advantage if called upon to operate on one single variable. *Alphacode* was designed to fulfil this purpose and, like T.I.P., the intention was to devise a code which would attract the reluctant engineer or scientist to become a computer user. The considerations which influenced the choice of order code were therefore (i) the necessity to remove the jargon so frequently associated with autocodes—all instructions were to be capable of expression in everyday language; (ii) the necessity to eliminate any reference to what goes on between presenting a program written in English and the printed set of results; (iii) the system was to be particularly suited to "one-off" problems: that is, time of operation could take second place to ease of programming and testing; (iv) the order code was to be easy to learn and use, and was to be such that a program could easily have extra instructions inserted in it; (v) counting and instruction modification should be simple, and the necessity for resetting of counters and modified instructions should be avoided as these are unnatural operations to one accustomed to working with a slide-rule or desk machine. The order code has 64 possible functions, including such comprehensive ones as "Solve Differential Equations," "Sum Power Series," "Interpolate in Graphical Data" and "Integrate by Simpson's Rule." The resulting order code is, in many respects, similar to those devised by Brooker (1958). The problem of writing in English, and the greater problem of having a neat and tidy record of a program after it has been tested and altered (a rare thing among programmers) was solved by maintaining a pulling file of pre-punched cards with all possible orders, on to which only the addresses have to be punched. It was in any case necessary to have only one order per card, in order to give the greatest flexibility for inserting or re-arranging orders, and this allowed enough room for a comprehensive definition of the order. Thus any tested program can immediately be listed and a fair copy, devoid of all jargon, can be produced.

The instruction word is of the form, "$a = b$ function c" where, for a function of one variable, the b is omitted. The variables, which are stored as fully floating numbers, are denoted by the letter x and a suffix, so that typical instructions are:

```
      x3 =  x4  ×        x5
7     x5 =      log      x3
      x6 =      sinh     x5
Print 12 results         from x1 onwards.
```

63 stores, referred to as N_1 to N_{63}, are reserved for counting in integers, but simple arithmetical operations may be carried out on the contents. The instruction for going round a loop a number of times is of the form

Count N_1 up to N_2 jumping to R7

which has the following effect. One is added to the counter N_1, which is then compared with N_2. If $N_1 \neq N_2$ the program jumps to the instruction which bears a reference number 7, as in the second typical instruction above. When N_1 and N_2 are equal, N_1 is automatically reset to zero and the program continues normally. References in this form are used for all discriminations and unconditional jumps.

6 GENERAL REMARKS

The three interpretive schemes for DEUCE described here, though not the only ones, have been most widely used. They form a mutually complementary set in that they cover a wide spectrum of scientific problems, and can be easily used by the beginner with little or no previous experience, without the need for a course in programming. Being interpretive rather than translation programs, they are not as fast as orthodox programs though, with G.I.P., the speed is virtually that of an orthodox program if the data is bulky enough. The other two schemes, which require less knowledge of DEUCE, are designed with an eye to "one-off" problems. Current developments in this field are tending more to meet the requirement of full translation from a pseudo-code to an orthodox DEUCE program. At a recent symposium* several speakers referred to the difficulty of such procedures on medium-size machines with two or more levels of storage, but the problems are by no means incapable of solution.

* Held in London by the British Computer Society on 1 August 1958.

7 ACKNOWLEDGEMENTS

This paper describes work performed on DEUCE by a number of people, including particularly the staffs of the National Physical Laboratory and Bristol Aero Engines Limited, to whom the author gratefully offers his acknowledgements. The paper is published by permission of the Director of Research, Nelson Research Laboratories, English Electric Co. Ltd.

REFERENCES

(1) GILMOUR, A. (1958). "The Solution of Railway Problems on a Digital Computer," *The Computer Journal*, Vol. 1, p. 25.
(2) BROOKER, R. A. (1958). "The Autocode Programs developed for the Manchester University Computers," *The Computer Journal*, Vol. 1, p. 15.

A New Programming Technique for Rational Fractions

by K. M. Howell

Summary: A method of programming calculations with rational fractions in terms of their prime factors is described. Three storage locations are used to accommodate the exponents of the prime factors up to 103, with an additional location for factors larger than this, or for a zero factor. The application of the technique to calculating Wigner 6j-symbols is outlined.

INTRODUCTION

A subroutine is available in the library for the Ferranti Pegasus computer (Ferranti Ltd., 1957), which carries out arithmetic operations on rational fractions with numerator and denominator stored separately. The technique described in this paper was developed independently and differs markedly from the library routine. It is particularly suitable for evaluating formulae which involve factorials and, in the case for which it was designed, it is probable that for some of the functions the library program would not have allowed for a sufficiently wide range of intermediate results, even though the final results are within range.

DEFINITION OF THE WIGNER 6j-SYMBOL

For some of the work being carried out at Southampton University in the field of theoretical nuclear physics, it was thought desirable to have a definitive table of Racah coefficients (Racah, 1942) in order to check and augment those already in existence (Obi, Ishidzu, Horie, Yanagawa, Tanabe and Sato, 1953 and 1954), (Sharp, Kennedy, Sears and Hoyle, 1953). The most convenient function to calculate was the Wigner 6j-symbol which differs from the Racah coefficient only by a phase factor and is more symmetrical in the six arguments:

In this expression, only factorials of numbers occur. In the method of calculation employed, all numbers were stored in terms of their prime factors. A table of factorials in this form was provided in the routine. For example:

$$53! = 2^{49}.3^{23}.5^{12}.7^{8}.11^{4}.13^{4}.17^{3}.19^{2}.23^{2}.29.31.37.41.43.47.53$$

All that is stored is the set of exponents, 49, 23, 12, 8, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 0, 0, etc. It was decided that primes up to about 100 would suffice, and that the counts (exponents) could be packed into three words in the computer. Obviously the count will be larger for the small primes than for the large, and so more digits are needed to contain them. Pegasus has a word-length of 38 binary digits plus a sign digit, 39 in all. The following allotment of the 114 useful digits in three words was made:

8 digits for the factor 2
7 digits for the factor 3
6 digits each for the factors 5, 7, 11
5 digits each for the factors 13, 17, 19
4 digits each for the factors 23 to 59
3 digits each for the factors 61 to 103.

$$\begin{Bmatrix} a & b & e \\ d & c & f \end{Bmatrix} = \left[\frac{(a-b-e)!(a-b-e)!(-a-b-e)!(a-c-f)!(a-c-f)!(-a-c-f)!}{(a-b-e-1)!\qquad(a+c-f-1)!}\right]^{1/2}$$
$$\cdot\left[\frac{(b-d-f)!(b-d+f)!(-b-d-f)!(c+d-e)!(c-d+e)!(-c-d-e)!}{(b+d-f-1)!\qquad(c+d-e+1)!}\right]^{1/2}$$
$$\cdot\sum_{t}\frac{(-1)^{t}(t+1)!}{(t-a-b-e)!(t-a-c-f)!(t-b-d-f)!(t-c-d-e)!(b+c+e+f-t)!(a-b-c-d-t)!(a-d-e-f-t)!}.$$